Use of Aliasing in SystemVerilog Verification Environment

Evean Qin
Advanced Micro Devices, Inc.
1 Commerce Valley Drive East, Markham, ON, Canada

Introduction

Building a state-of-art verification environment for a design is essentially building a large complex software system. Many constructs and techniques from computer programming are supported by the verification language for convenience in development. However, some commonly used ones are not. For instance, "***alias***" is not defined in the SystemVerilog Standard. Despite the reasons regarding not having "alias" in the language, in some cases, aliasing can be very handy for designing an efficient testbench. It helps to reduce the maintenance effort and prevents human mistakes while building or using the verification environment.

Example of a Design and Testbench

The DUT contains a command interface which consists of a 16-bit opcode bus and a 32-bit operand bus. In this command bus, according to the Opcode, the Operand can be dynamically divided into different fields.

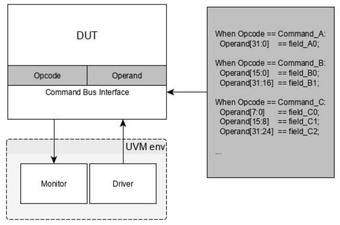

Figure 1. Design and Testbench Example

For example, when the Opcode is ***COMMAND_A***, the Operand will be assigned to a 32-bit field: ***fields_A0***. When the Opcode is ***COMMAND_B***, the Operand will be divided into two 16-bit fields instead: ***field_B0*** and ***field_B1***. Sometimes, the Operand does not have to be fully filled and the fields do not have to be consecutive. For example, when the Opcode is ***COMMAND_C***, the Operand is setup to have ***three 8-bit fields*** with ***8 reserved bits*** between two of them.

The UVM Environment Setup

In the UVM verification environment, a monitor and a driver shall be set up for this interface. A base transaction is designed to contain an *ENUM* member "opcode" associated with the Opcode bus, and a 32-bit data member "operand" associated with the Operand bus. The driver and monitor operate with this base transaction. And multiple child transactions are extended from this base for different commands, and ought to store their own unique fields according to the specification. The UVM tests execute the testing sequences with different command transactions based on the test intents.

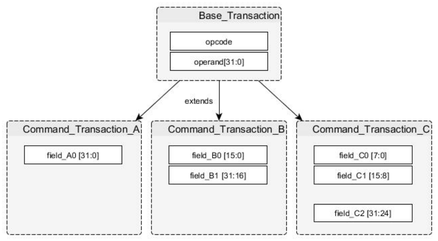

Figure 2. Example of the Transaction Hierarchy Setup

Problems in the Design

Here are the two typical problems when designing and using the UVM environment:

a) Extra actions for packing and unpacking the operands. *For example, the driver needs to decode the opcode then pack the fields from the command transaction into the operand before sending it into the interface. Vice versa for the Monitor.*

b) Data synchronization between variables is not seamless. *For example, in the command transaction class, the same information is stored in two independent sets of variables. Updating either of them does not imply updating the other one.*

Usually, in a programming language, the above problems can be solved by aliasing the variables, so that either the operand or any field is updated in the command transaction, the value will be synchronized without any extra action. Here are the pseudo codes in Command_Transaction_A:

```
class Command_Transaction_A extends Base_Transaction;
  …
  alias field_A0 = operand[31:0];
  …
```

As mentioned previously, "***alias***" is not allowed in a class construct in SystemVerilog. The pseudo codes will fail to be compiled. Therefore, we will start the journey of seeking for the solutions to workaround this and hopefully achieve the desired goal in the testbench design.

Potential Alternative Solution

A. Standard-Defined "alias"

SystemVerilog 3.1 has introduced "***alias***" into the standard. The alias statement models a bidirectional short-circuit connection used in a module or an interface construct, by which the designer can assign different "virtual" nets sharing the same physical net. Here is a code example of the "***alias***" usage in an interface:

```
interface Command_A_inf (inout wire [31:0] operand);
  wire [31:0] field_A0;
  alias field_A0 = operand;
  …
endinterface
```

However, each command needs its own virtual interface to the bus for the monitor and drive. Adding a new command, a new interface needs to be created. Using the standard-defined "***alias***" statement does not provide any convenience in the verification environment.

B. let Construct

SystemVerilog from the *IEEE 1800-2012* standard introduces the "***let***" construct, which defines a template expression, customized by its ports. Besides customizing the text macros, it can be used to provide shortcuts for identifier. It can be used in UVM by declaring it in a SystemVerilog package.

However, the let identifier cannot be used in the left-hand-side (LHS) of an assignment as shown in the example below. So the user will not be able to update the operand through assigning values directly to the fields in the command transactions. Here is an example:

```
package command_bus_dv_package;
  bit [31:0] operand;
  let field_A0 = operand;
endpackage
class Command_Transaction_A extends Base_Transaction;
  function set_A0(bit[31:0] value);
    field_A0 = value;  //This is illegal
  endfunction
endclass
```

Furthermore, the ***let*** construct is not supported in a class construct, and hence, all its declarations have to be put inside a package. This may cause name conflicts and data corruptions due to multiple concurrent assignments. Therefore, the ***let*** construct seems unsuitable for linking the variable bi-directionally.

C. Packed Union Struct

With *SystemVerilog 3.1*, packed union can be defined to concatenate multiple packed or integer data into a packed array. It allows the users to access the union as a 32-bit data type and its members independently. Most importantly, it can be used in a package or a class construct. Here is an example:

```
typedef union packed {
  bit[15:0] field_B0;
  bit[15:0] field_B1;
} commandB_union;
  …
```

When using this setup, different packed union types need to be defined for different command transactions. The operand variable in the base transaction is decoupled with the operand variable in its child transactions due to the data type overriding. As a result, the driver cannot directly drive the Operand bus with the value from the "operand" in the base transaction. In the end, this testbench structure with additional static components is not dynamic enough to solve the problem but increases work in maintenance.

D. Dynamic Methods Lookup

In the SystemVerilog standard from *IEEE 1800-2012*, dynamic methods lookup is introduced. It allows the use of a variable of the superclass type to hold subclass objects and to reference the method of those subclasses directly from the superclass variable. In practice, the pure virtual methods can be declared in the Base_Transaction for packing and unpacking the operand. The actual method definitions are implemented in the child classes such as Command_Transaction_B. Here is a code example:

```
class Base_Transaction;
  …
  pure virtual function void pack_operand();
  pure virtual function void unpack_operand();
  …
endclass
```

```
class Command_Transaction_B extends …;
  //Define the virtual functions here
  virtual function bit[31:0] pack_operand();
    operand = {field_B1, field_B0};
    return operand;
  endfunction
  virtual function void unpack_operand();
    field_B0 = operand[15:0];
    field_B1 = operand[31:16];
  endfunction
endclass
```

This polymorphism technique eliminates the casting actions, which saves some code and solves problems of bit-packing in the testbench development. However, it still relies on the functions to synchronize the data between variables. Therefore, the users and the testbench designers need to carefully decide when and where to call these functions to avoid data corruption.

The User-Defined Aliasing Method

After exhausting the potential solutions from the SystemVerilog Standard, it is concluded that none of them can fully resolve the problems in the testbench construction for this design. Therefore, let's come back to the aliasing method and see if this can be implemented in a class construct for the verification environment.

In a computer system, aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program. Hence, implementing a user-defined alias in a SystemVerilog class construct needs to retrieve the memory address of the variable first. But SystemVerilog does not define pointer type and has no system function to retrieve the memory address of a variable either. In this case, to leverage the fine establishment of the pointer referencing in C/C++, the DPI-C plugin can be used to work with SystemVerilog for this problem.

Here is an example of the DPI-C functions and the usage in SystemVerilog environment:

```
#include <stdio.h>
#include "vc_hdrs.h"

int get_pAddress(int* variable) {
  return variable;
}
int get_pValue(int address){
  int *addr;
  addr = address;
  return *addr;
}
void set_pValue(int address, int value){
  int *addr;
  addr = address;
  *addr = value;
}
```

```
class VARIABLE #(int WIDTH=32);
  rand bit[WIDTH-1:0] value;
  int offset;
  …
  function int get_value();
    value = get_pValue(pointer + offset);
    return value;
  endfunction
  function void set_value(bit[WIDTH-1:0]
_value);
    bit[31:0] full_value = get_pValue(address);
    bit[31:0] mask = {WIDTH{1'b1}};
    full_value = full_value &
(~(mask<<(offset*8))) | (_value<<(offset*8));
    value = _value;
    set_pValue(pointer, full_value);
  endfunction
  function void set_alias(VARIABLE _alias, int
_offset=0);
    pointer = _alias.pointer;
    offset = _offset;
    value = get_pValue(pointer + offset);
  endfunction
endclass
```

Here is an example of how to declare and use the alias:

```
class Command_Transaction_B extends Base_Transaction
  VARIABLE#(16) field_B0;  // 16-bit field
  VARIABLE#(16) field_B1;  // 16-bit field
  …
  field_B0.set_alias(operand, 0);
  field_B1.set_alias(operand, 2);
  …
endclass

class example_test;
  Command_Transaction_B trans;
  …
  trans.field_B0.set_value(16'hbbbb);
  trans.field_B1.set_value(16'h2222);
  // This will print "trans.operand = 0x2222bbbb"
  `uvm_info("EXAMPLE", $sformatf("trans.operand = 0x%x", trans.operand.get_value()), …)
endclass
```

Summary

Though the DPI-C approach has some solvable limitations, it empowers the verification environment with the "***alias***" method. Different variables can be linked dynamically to reduce the effort in testbench development and maintenance. With an insignificant run-time overhead, the verification environment can become more scalable, less design error-prone, and more user friendly.